02 NFAs and Regular Language Closures

Ryan Bernstein

1 Introductory Remarks

- Again, for anybody who wasn't here last time, homework 1 is available on the course web site. It's due on Thursday

1.1 Recapitulation

Last lecture, we introduced the deterministic finite automaton, which was the simplest type of abstract machine that we'll be constructing in this class. Like all of the machines we'll be looking at, DFAs take strings as input and either accept or reject them depending on whether or not those strings are in the language of the machine.

DFAs are drawn as a collection of states, some of which are "accepting" or "final" states, and transitions between those states. Computation in a DFA begins at the start state, and we follow a transition to another state on each character in the input string.

We need to formalize a DFA if we want to represent it in a computer. To do so, we say that a DFA is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a set of states
- Σ is the alphabet of the language that the machine decides
- $\delta : (Q \times \Sigma) \to Q$ is the transition function. It takes a state/input pair and returns another state
- q_0 is the start state. $q_0 \in Q$.
- F is the set of accepting states. $F \subseteq Q$.

2 DFA Wrap-Up

2.1 Worksheet Problem Solution

At the end of the last lecture, we looked at a worksheet problem wherein we had to prove that regular languages were closed under complement — that is to say, if A is a regular language, then $\overline{A}$ is also a regular language. This was an example of a proof by construction.

Last lecture, we'd established the following definition of a regular language:

> A language is *regular* if and only if there is a DFA that decides it.

In CS 251, we learned that "if and only if" can be thought of as a bidirectional implication. That is to say:

- If a DFA decides A, then A is a regular language
- If A is a regular language, then there is a DFA that decides it.

The second part is the key to this particular proof.

If A is a regular language, then there exists a DFA $D = (Q, \Sigma, \delta, q_0, F)$ that decides it. To show that $\overline{A}$ is also regular, we need to construct a machine D' that decides $\overline{A}$. Since $\overline{A}$ is the language of strings that D rejects, we can construct D' as follows:

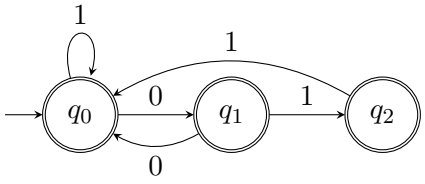
$$D' = (Q, \Sigma, \delta, q_0, Q - F)$$

In other words, since membership is determined by whether or not the last state is in F, we can create a machine that accepts the strings rejected by D by taking our existing machine and using $\overline{F}$ (in the universe Q) the set of final states.

Because we can transform a DFA that decides an arbitrary regular language A into a machine that decides $\overline{A}$, we can say that the complement of any regular language is also regular.

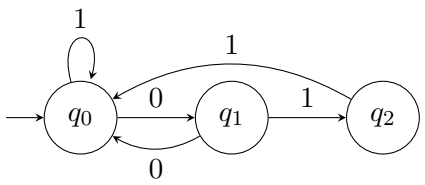
2.2 Fully-Specified Transition Functions

In last lecture's discussion, we mandated that to be a DFA, our machines had to have fully-specified transition functions — that is to say, every possible state/character pair should have a transition defined for it. This transition is a great illustration of why. Considering the following machine, which accepts strings in $\{0, 1\}^*$ that do not contain the substring 010:



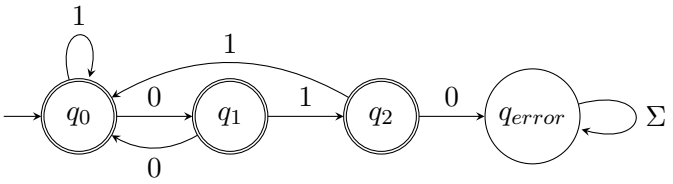
This machine doesn't contain a fully-specified transition function, since it doesn't define what to do when we see a 0 from q_2. A reasonable interpretation might be that we should crash on any state/input pair for which no transition is defined.

If we did this, though, then using the above logic to try and construct a machine that decides $\overline{L(M)}$ would create the following machine:



This machine *should* be accepting any string that *does* contain the substring 010. However, it no longer has any accepting states, which means that the language of this machine is $\emptyset$, not $\overline{L(M)}$.

If we mandated that the transition function be fully specified, our original machine would probably have looked something like this:



Replacing F with $\overline{F}$ now turns this last state from an inescapable failure to an inescapable success, and the machine decides $\overline{L(M)}$ as expected.

If we treat unspecified transitions as auto-rejecting "crashes", it's easy to see that mandating a full specification doesn't stop us from creating a DFA for any language. If we had some machine containing undefined transitions, we can easily make a fully-specified machine by:

- Adding a new state q_{error} to Q
- Adding a transition to q_{error} on any state/input pair for which no transition is already specified
- Adding a loop transition to q_{error} for any state/input pair for which q_{error} is already the current state

Allowing DFAs to leave out information would make constructive proofs more difficult, since without knowing what information was included in a machine — about which we may know nothing, as in the complement proof — we wouldn't know what information we could use to construct a new machine.

Constructing fully-specified machines can get tedious, though. And the example that we used in our justification for full specification brings to mind an excellent point: if we can build any machine from which we can *construct* a fully-specified machine without ambiguity, then we can use that "shorthand" machine in place of the fully-specified machine in any proof (or even any implementation).

3 Nondeterministic Finite Automata

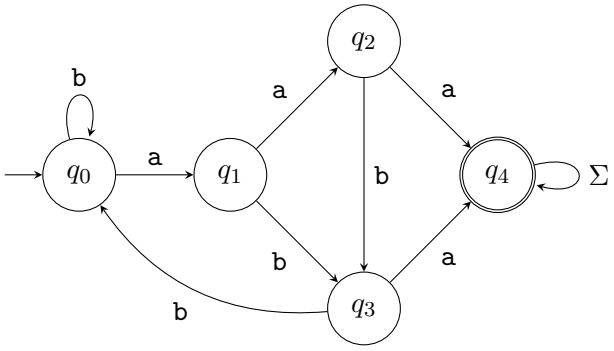
These shorthand machines are known as *nondeterministic finite automata*, or NFAs. One of the things that separates them from DFAs is exactly what we've been talking about: an NFA need not specify a mapping for every possible state/transition pair.

3.1 Multiply-defined Transitions

Our definition of a deterministic function was that we had at most one output for any given input. The determinism in our deterministic finite automata came from our requirement that the transition function be deterministic.

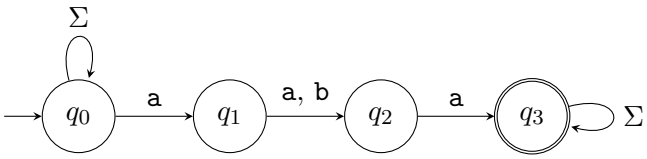
It makes sense, then, for us to relax this requirement for an automaton that we call nondeterministic. In an NFA, it's perfectly valid to have multiple transitions defined for the same input from the same current state. Why is this useful? Let's look at an example.

Let's say we're trying to show that $L = \{s \in \{\texttt{a}, \texttt{b}\}^* \mid s \text{ contains the substring } \texttt{aaa} \text{ or } \texttt{aba}\}$ is regular. To construct a DFA for this language, we'd have to do something like this:



It's easy to see that DFAs like this can balloon very quickly. Along with adding branching states for single-character differences in substrings, we also have to consider transitions between those branches each time we see a character that we weren't expecting.

If we can define multiple transitions for the same state/input pair, though, then we can construct an NFA to recognize this language like this:



Since we have the option of looping back to the start state on every character, we can think of this machine as searching for the substrings in question by iterating over the string's characters c_1 to $c_{|s|}$ and seeing if either substring starts at c_i. This approach, coupled with the fact that we don't have to include a transition for every state/input pair, means that we no longer have to worry about transitioning back to intermediate states (or the start state) if we see an input that isn't part of one substring.

This is pretty similar to a simple algorithm that we might use if we had to write a function that decided this same question. In Python, such a function might be written like this:

```
def contains(string, substring):
  for i in range(0, len(string) - len(substring)):
    for j in range(len(substring)):
      if string[i + j] != substring[j]:
        continue
      return true

  return false
```

Of course, this Python implementation is doing things that a state machine can't, since we're looking ahead in the string, but the idea is similar.

The difference in the number of states is pretty small, but the difference in complexity is, in my opinion, pretty immediately visually apparent.

3.1.1 Complications

Of course, having multiple transitions defined for the same state/input pair causes some complications. How do we know which branch is "correct"?

The truth is, we don't. So we take both. When we're in a state that contains multiple transitions for the next character, we pass the next character to *all* such transitions and run the remainder of the string through all of the new states "in parallel".

In a DFA, we took for granted the idea that our current status was some single element of Q. In a DFA, the current computational context — which we'll call C for the moment — is instead a *set* of elements of Q. When we see a new input character x, we iterate over each $q \in C$ and replace it with $\delta(q, x)$.

In a DFA, δ was a function from $Q \times \Sigma$ to Q. Since we still write transitions as coming from each state individually, the domain of δ in an NFA is still $Q \times \Sigma$. But now, since we produce *zero* (for an undefined transition) *or more* next states, the range of δ is $\mathbb{P}(Q)$.

Let's walk through the series of states that this machine traverses while examining the string *aabbaaa*:

Remaining Input	States
`aabbaaa`	$\{q_0\}$
`abbaaa`	$\{q_0, q_1$
`bbaaa`	$\{q_0, q_2\}$
`baaa`	$\{q_0\}$
`aaa`	$\{q_0\}$
`aa`	$\{q_0, q_1\}$
`a`	$\{q_0, q_1, q_2$
ϵ	$\{q_0, q_1, q_2, q_3\}$

Note that since our current computational context is a *set* of states, we don't have multiple copies of a state in context, even when we reach the same state via multiple transitions. If we see a state/character input pair for which no transition is defined, we simply remove the offending element from context.

3.1.2 Acceptance Criteria

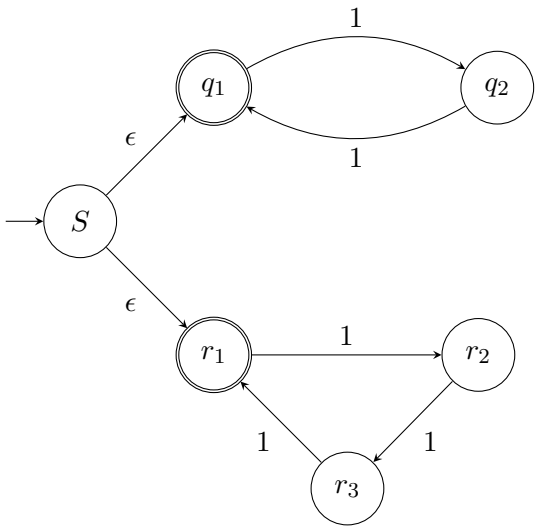
If our current context consists of multiple states, how do we determine whether an NFA accepts or rejects a string? For this, we can look back at our previous view of having one "correct" path that we follow to completion, rather than following all multiply-defined transitions at once.

An NFA accepts a string s if *any* of the states in the computational context at the end of the string are elements of F; it rejects s if none of them are elements of F.

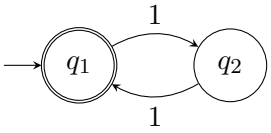
3.2 Epsilon Transitions

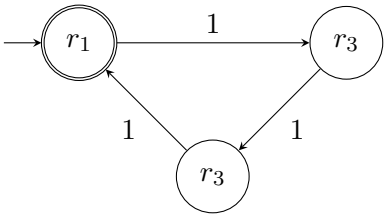
NFAs can do one other thing that DFAs can't: they can change state without an associated change in input. Transitions that don't affect the input stream are known as ϵ-transitions. We can think of this quite literally: they're transitions that we follow when removing the empty string ϵ from the input.

When are these useful? We often use ϵ-transitions when combining multiple machines to recognize some combination of languages. For instance, consider the following example:



This machine decides $L = \{1^k \mid k \text{ is even or a multiple of three}\}$. We can consider this as a combination of two "submachines" that decide these cases individually:

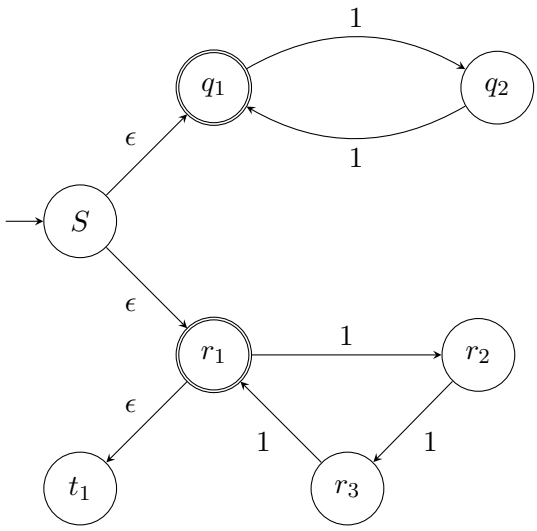




3.2.1 The ϵ-Closure

To address the way that ϵ transitions alter the behavior of NFAs, we'll now introduce the idea of an ϵ-closure. The ϵ-closure of a state q is the set containing q and any state that can be reached from q via nothing but ϵ-transitions.

Currently, the ϵ-closure of S is $\{S, q_1, r_1\}$. Let's say now that we added another ϵ-transition from r_1 to some new state, t_1, like so:



Now, the ϵ-closure of S is $\{S, q_1, r_1, t_1\}$ — the ϵ-closure extends across multiple ϵ-transitions. The ϵ-closure of r_1 is $\{r_1, t_1\}$. Since r_1 is in the ϵ-closure of S, its ϵ-closure is a subset of the ϵ-closure of S.

3.2.2 Approximating ϵ-Transitions with Nondeterminism

We'll now show that ϵ-transitions don't allow NFAs to decide languages that they wouldn't be able to without them. To do this, we'll provide an algorithm to remove ϵ-transitions from an NFA without changing the language that the NFA decides.

Proof by Construction Let N be an NFA. We can construct an NFA N' without ϵ-transitions as follows:

For each state $p \in Q$:

1. Make p an accepting state of N' if and only if the ϵ-closure of p contains an accepting state of N
2. Add a transition from $p \to q$ labeled x if and only if there is a transition labeled x from some state in the ϵ-closure of p to q.

Delete all ϵ-transitions.

3.3 Worksheet Exercise: Building NFAs

3.4 Formalizing an NFA

An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is still a set of states
- We have to add ϵ to our alphabet, so we can say that $\Sigma_{NFA} = \Sigma \cup \{\epsilon\}$. We often abbreviate this as Σ_ϵ.
- $\delta : Q \times \Sigma_\epsilon \to \mathbb{P}(Q)$ is a transition function

- q_0 is again the start state and an element of Q
- F is again a subset of Q consisting of final states

4 Languages that NFAs can Recognize

We introduced the NFA as a "shorthand" machine that we could use to unambiguously create a DFA the decided the same language. For this to be true, we need to show that NFAs and DFAs are equivalent in power — that is to say, that the set of languages that can be decided by an NFA is exactly the same as the set of languages that can be decided by a DFA (i.e. the regular languages).

Again, from CS 251, we can think of equivalence as a bidirectional implication and prove this in two separate steps:

1. Show that if there is a DFA that decides L, then there is also an NFA that decides L
2. Show that if there is an NFA that decides L, then there is also a DFA that decides L

4.1 Constructing an NFA from any DFA

Constructing an NFA from a DFA is trivial, since NFAs can do everything that DFAs can. A DFA can already be thought of an NFA that never utilizes any of its nondeterministic functionality.

4.2 Constructing a DFA from any NFA

Before we begin, I should emphasize that while this algorithm will create an equivalent DFA for any arbitrary NFA, there may be some other equivalent DFA that is more efficient (i.e. contains fewer states). We don't actually care about efficiency for the first three quarters of this class, so for the moment, this is sufficient.

What is the type of each element of Q? The truth is, we don't actually care. As long as the elements are unique — which they must be, since Q is a set — we simply draw a circle for each item. This is possible because the transitions are stored in δ, rather than associated with the states themselves.

What is the type of the current computational context in a DFA? Again, we don't care, so long as it is a single element of Q.

After removing ϵ transitions, we can therefore turn an NFA context into a DFA context by making every possible set of states in the NFA into a single state in the DFA context:

- $Q_D = \mathbb{P}(Q_N)$
- $\Sigma_D = \Sigma_N$
- $\delta_D(R, x) = \{q \in \mathbb{P}(Q_N) \mid \delta_N(r, x) = q \text{ for some } r \in R\}$
- $q_{0_D} = \{q_{0_N}\}$
- $F_D = \{S \in \mathbb{P}(Q_N) \mid S \text{ contains any element of } F_N\}$

4.3 Worksheet Exercise: NFA to DFA Conversion

4.4 The Result: A New Definition of Regular Languages

Since we've shown that NFAs and DFAs are equivalent in power, we can now relax our definition of regular languages slightly. Before, we said that a language was regular if and only if there was a deterministic finite automaton that decided it. We can now simply say:

A language is regular if and only if there is a finite automaton that decides it.

5 Closure Properties and the Regular Operations

We'll now prove some more closure properties of the regular languages. While we could have done this using DFAs, it's much simpler using nondeterminism.

5.1 The Regular Languages are Closed under Union

This is a generalization of the procedure that we used to decide $\{1^k \mid k \text{ is even or a multiple of } 3\}$ earlier.

Assume that A and B are regular languages. Then there are DFAs, M_A and M_B, that decide them.

We can construct an NFA N that decides $A \cup B$ by adding a new start state, q_s, and adding an ϵ transition from q_s to the start states of M_A and M_B.

5.2 Worksheet Exercise: The Regular Languages are Closed under Concatenation

5.3 The Regular Languages are Closed under the Kleene Closure

Let A be a regular language. Then there is a DFA D that decides it.

We can construct an NFA that decides A^* as follows:

1. Add a new start state, q_s
2. Add an ϵ-transition from q_s to the original q_0
3. Add an ϵ-transition from each accepting state to q_s

5.4 The Regular Operations

We've seen now that the regular languages are closed under union, concatenation, and Kleene star. Because of this fact, these operations are known as the *regular operations*.

This gives us a new way to prove regularity of a language: if I can show that a language L can be created by composing regular operations over other known (or trivially) regular languages, then L itself is also regular. I don't *have* to construct a DFA or NFA that decides L, because I already know that I can.